Expressiveness via Intensionality and Concurrency

Thomas Given-Wilson

Expressiveness via Intensionality and Concurrency

Thomas Given-Wilson ⋆

INRIA, Paris, France
thomas.given-wilson@inria.fr

Abstract. Computation can be considered by taking into account two dimensions: extensional versus intensional, and sequential versus concurrent. Traditionally sequential extensional computation can be captured by the λ-calculus. However, recent work shows that there are more expressive intensional calculi such as SF-calculus. Traditionally process calculi capture computation by encoding the λ-calculus, such as in the π-calculus. Following this increased expressiveness via intensionality, other recent work has shown that concurrent pattern calculus is more expressive than π-calculus. This paper formalises the relative expressiveness of all four of these calculi by placing them on a square whose edges are irreversible encodings. This square is representative of a more general result: that expressiveness increases with both intensionality and concurrency.

1 Introduction

Computation can be characterised in two dimensions: *extensional* versus *intensional*; and *sequential* versus *concurrent*. Extensional sequential computation models are those whose *functions* cannot distinguish the internal structure of their *arguments*, here characterised by the λ-calculus [3]. However, Jay & Given-Wilson show that λ-calculus does not support all sequential computation [19]. In particular, there are intensional Turing-computable functions, characterised by *pattern-matching*, that can be represented within SF-calculus but not within λ-calculus [19]. Of course λ-calculus can encode Turing computation, but this is a weaker claim. Ever since Milner et al. showed that the π-calculus generalises λ-calculus [23, 25], concurrency theorists expect process calculi to subsume sequential computation as represented by λ-calculus [23, 25, 24]. Following from this, here extensional concurrent computation is characterised by process calculi that do not communicate terms with internal structure, and, at least, support λ-calculus. Intensional concurrent computation is represented by process calculi whose communication includes terms with internal structure, and reductions that depend upon the internal structure of terms. Here intensional concurrent computation is demonstrated by *concurrent pattern calculus* (CPC) that not only generalises intensional pattern-matching from sequential computation to *pattern-unification* in a process calculus, but also increases the *symmetry* of interaction [13, 14].

These four calculi form the corners of a *computation square*

$$\begin{tikzcd}
\lambda_v\text{-calculus} \arrow[r] \arrow[d, dashed] & SF\text{-calculus} \arrow[d, dashed] \\
\pi\text{-calculus} \arrow[r] & \text{concurrent pattern calculus}
\end{tikzcd}$$

⋆ This work has been partially supported by the project ANR-12-IS02-001 PACE.

where the left side is merely extensional and the right side also intensional; the top edge is sequential and the bottom edge concurrent. All the arrows preserve reduction. The horizontal (solid) arrows are *homomorphisms* in the sense that they also preserve *application* or *parallel composition*. The vertical (dashed) arrows are *parallel encodings* in that they map application to a parallel composition (with some machinery). Thus each arrow represents increased expressive power with CPC completing the square.

This paper presents the formalisation of these expressiveness results for the four calculi above. This involves adapting some popular definitions of encodings [15–17] and then building upon various prior results [8, 23, 25, 13, 19, 11]. These can be combined to yield the new expressiveness results here captured by the computation square.

The organisation of the paper is as follows. Section 2 reviews prior definitions of encodings and defines the ones used in this paper. Section 3 reviews λ-calculus and combinatory logic while introducing common definitions. Section 4 summarises intensionality in the sequential setting and formalises the arrow across the top of the square. Section 5 begins concurrency through π-calculus and its parallel encoding of λ_v-calculus. Section 6 recalls concurrent pattern calculus and completes the results of the computation square. Section 7 draws conclusions, considers related work, and discusses future work.

2 Encodings

This section recalls valid encodings [17] for formally relating process calculi and adapts the definition to define homomorphisms and parallel encodings. The validity of valid encodings in developing expressiveness studies emerges from the various works [15–17], that have also recently inspired similar works [21, 22, 30]. Here the adaptations are precise definitions of homomorphisms that give stronger positive results (the negative results are not required to be as strong). The parallel encodings are defined to account for the mixture of sequential and concurrent languages considered.

An *encoding* of a language $\mathcal{L}_1$ into another language $\mathcal{L}_2$ is a pair $([\![\cdot]\!], \varphi_{[\![\,]\!]})$ where $[\![\cdot]\!]$ translates every $\mathcal{L}_1$-term into an $\mathcal{L}_2$-term and $\varphi_{[\![\,]\!]}$ maps every name (of the source language) into a tuple of k names (of the target language), for $k > 0$. The translation $[\![\cdot]\!]$ turns every term of the source language into a term of the target; in doing this, the translation may fix some names to play a precise rôle or may translate a single name into a tuple of names. This can be obtained by exploiting $\varphi_{[\![\,]\!]}$.

Now consider only encodings that satisfy the following properties. Let a *k-ary context* $C(\_{}_1;\ldots;\_{}_k)$ be a term with k holes $\{\_{}_1;\ldots;\_{}_k\}$ that appear exactly once each. Moreover, denote with $\longmapsto_i$ and $\Longmapsto_i$ the relations $\longmapsto$ (reduction relation) and $\Longmapsto$ (the reflexive transitive closure of $\longmapsto$) in language $\mathcal{L}_i$; denote with $\longmapsto_i^\omega$ an infinite sequence of reductions in $\mathcal{L}_i$. Moreover, let $\equiv_i$ denote the structural equivalence relation for a language $\mathcal{L}_i$, and $\sim_i$ denote a strong reference behavioural equivalence for language $\mathcal{L}_i$ and $\simeq_i$ a weak reference behavioural equivalence. For simplicity the notation $T \longmapsto_i \equiv_i T'$ denotes that there exists T'' such that $T \longmapsto_i T''$ and $T'' \equiv_i T'$, and may also be used with $\Longmapsto_i$ or $\sim_i$ or $\simeq_i$. Also, let $P \Downarrow_i$ mean that there exists P' such that $P \Longmapsto_i P'$ and $P' \equiv_i P'' \mid \surd$, for some P'' where $\surd$ is a specific process to indicate success. Finally, to simplify reading, let S range over terms of the source language (viz., $\mathcal{L}_1$) and T range over terms of the target language (viz., $\mathcal{L}_2$).

Definition 1 (Valid Encoding (from [17])). *An encoding* $(\llbracket \cdot \rrbracket, \varphi_{\llbracket\,\rrbracket})$ *of* $\mathcal{L}_1$ *into* $\mathcal{L}_2$ *is* valid *if it satisfies the following five properties:*

1. Compositionality: *for every k-ary operator* op *of* $\mathcal{L}_1$ *and for every subset of names* N*, there exists a k-ary context* $C^N_{\mathsf{op}}(\_{}_1; \ldots ; \_{}_k)$ *of* $\mathcal{L}_2$ *such that, for all* $S_1, \ldots, S_k$ *with* $\mathsf{fn}(S_1, \ldots, S_k) = N$*, it holds that* $\llbracket\, \mathsf{op}(S_1, \ldots, S_k) \,\rrbracket = C^N_{\mathsf{op}}(\llbracket\, S_1 \,\rrbracket; \ldots ; \llbracket\, S_k \,\rrbracket)$.
2. Name invariance: *for every* S *and name substitution* σ*, it holds that*
$$\llbracket\, \sigma S \,\rrbracket \begin{cases} = \ \sigma'\llbracket\, S \,\rrbracket & \text{if } \sigma \text{ is injective} \\ \sim_2 \ \sigma'\llbracket\, S \,\rrbracket & \text{otherwise} \end{cases}$$
where σ' *is such that* $\varphi_{\llbracket\,\rrbracket}(\sigma(a)) = \sigma'(\varphi_{\llbracket\,\rrbracket}(a))$ *for every name* a.
3. Operational correspondence:
   – *for all* $S \Longmapsto_1 S'$*, it holds that* $\llbracket\, S \,\rrbracket \Longmapsto_2 \sim_2 \llbracket\, S' \,\rrbracket$*;*
   – *for all* $\llbracket\, S \,\rrbracket \Longmapsto_2 T$*, there exists* S' *such that* $S \Longmapsto_1 S'$ *and* $T \Longmapsto_2 \sim_2 \llbracket\, S' \,\rrbracket$.
4. Divergence reflection: *for every* S *such that* $\llbracket\, S \,\rrbracket \longmapsto^{\omega}_2$*, it holds that* $S \longmapsto^{\omega}_1$.
5. Success sensitiveness: *for every* S*, it holds that* $S \Downarrow_1$ *if and only if* $\llbracket\, S \,\rrbracket \Downarrow_2$.

Observe that the definition of valid encoding is very general and, with the exception of success sensitiveness, can apply to sequential languages such as λ-calculus as well as process calculi. However, the relations presented in this work bring together a variety of prior results and account for them in a stronger and more uniform manner. To this end, the following definitions support the results. The first two define homomorphism in the sequential and concurrent settings.

Definition 2 (Homomorphism (Sequential)). *A* (sequential) homomorphism *is a translation* $\llbracket \cdot \rrbracket$ *from one language to another that preserves reduction and application.*

Definition 3 (Homomorphism (Concurrent)). *A* (concurrent) homomorphism *is a valid encoding whose translation preserves parallel composition.*

The next is for encoding sequential languages into concurrent languages and is strongly influenced by the definition of a valid encoding. Observe that $\llbracket \cdot \rrbracket_c$ indicates an encoding from source terms to target terms that is parametrised by a name c.

Definition 4 (Parallel Encoding). *An encoding* $(\llbracket \cdot \rrbracket_c, \varphi_{\llbracket\,\rrbracket})$ *of* $\mathcal{L}_1$ *into* $\mathcal{L}_2$ *is a* parallel encoding *if it satisfies the first four properties of a valid encoding (compositionality, name invariance, operational correspondence (with* $\sim_i$ *replaced by* $\simeq_i$*), and divergence reflection) and the following additional property.*

5. Parallelisation: *The translation of the application* MN *is of the form* $\llbracket MN \rrbracket_c \overset{\text{def}}{=} (\nu n_1)(\nu n_2)(R \mid \llbracket M \rrbracket_{n1} \mid \llbracket N \rrbracket_{n2})$ *where* R *depends only upon* c *and* n_1 *and* n_2.

The weakening of the operational correspondence from strong behavioural equivalence $\sim$ to weak behavioural equivalence $\simeq$ is required to work with the existing results. This weakening can be alleviated by compromising on other properties, e.g. compositionality and parallelisation, and even strengthened to structural equivalence. However, since the results here are building on those already in the literature, the definition is adapted here with other approaches discussed later (in Section 6.2). Despite this weakening, the preservation of reduction and divergence is maintained.

Parallelisation is a restriction on the more general compositionality criteria. Here this ensures that in addition to compositionality, the translation must allow for independent reduction of the components of an application. As the shift from sequential to concurrent computation can exploit this to support parallel reductions, the definition of parallel encoding encourages more flexibility in reduction since components can be reduced independently.

The removal of the success sensitiveness property is for simplicity when using prior results. It is not difficult to include success sensitiveness, this involves adding the success primitive to the sequential languages and defining $S \Downarrow$, e.g. $S \Downarrow$ means that $S \longmapsto^* \surd$. However, this requires adding a test process Q_c to the definition of parallel encoding with success sensitiveness defined by: "for every S, it holds that $S \Downarrow_1$ if and only if $[\![S]\!]_c \mid Q_c \Downarrow_2$. However, since adding the success state $\surd$ to λ-calculus and combinatory logics would require redoing many existing results, it is easier to avoid the added complexity since no clarity or gain in significance is made by adding it.

Encodings from concurrent languages into sequential ones have not been defined specifically here since they prove impossible. The proof of these results relies merely on the requirement of reduction preservation or operational correspondence, and so shall be done on a case-by-case basis.

3 Sequential Extensional Computation

Both λ-calculus and traditional combinatory logic base reduction rules upon the application of a function to one or more arguments. Functions in both models are extensional in nature, that is a function does not have direct access to the internal structure of its arguments. Thus, functions that are extensionally equal are indistinguishable within either model even though they may have different normal forms.

The relationship between the λ-calculus and traditional combinatory logic is closer than sharing application-based reduction and extensionality. There is a homomorphism from call-by-value λ_v-calculus into any combinatory logic that supports the combinators S and K [8, 3]. There is also a homomorphism from traditional combinatory logic to a λ-calculus with more generous operational semantics [8, 3].

3.1 λ-calculus

The *term* syntax of the λ-calculus is given by

$$t ::= x \mid t\ t \mid \lambda x.t\ .$$

The *free variables* of a term are defined in the usual manner. A *substitution* σ is defined as a partial function from variables to terms. The *domain* of σ is denoted $\mathsf{dom}(\sigma)$; the free variables of σ, written $\mathsf{fv}(\sigma)$, is given by the union of the sets $\mathsf{fv}(\sigma x)$ where $x \in \mathsf{dom}(\sigma)$. The *variables* of σ, written $\mathsf{vars}(\sigma)$, are $\mathsf{dom}(\sigma) \cup \mathsf{fv}(\sigma)$. A substitution σ *avoids* a variable x (or collection of variables μ) if $x \notin \mathsf{vars}(\sigma)$ (respectively $\mu \cap \mathsf{vars}(\sigma) = \{\}$). Substitution *composition* is denoted $\sigma_2 \circ \sigma_1$ and indicates that $\sigma_2 \circ \sigma_1(t) = \sigma_2(\sigma_1 t)$. Note that all substitutions considered in this paper have finite domain. The application of a substitution σ to a term t is defined as usual, as is α-conversion $=_\alpha$.

There are several variations of the λ-calculus with different operational semantics. For construction of the computation square by exploiting the results of Milner et al. [23], it is necessary to choose an operation semantics, such as *call-by-value* λ_v-calculus or *lazy* λ_l-calculus. The choice here is to use call-by-value λ_v-calculus, although the results can be reproduced for lazy λ_l-calculus as well. In addition a more generous operation semantics for λ-calculus will be presented for later discussion and relations.

To formalise the reduction of call-by-value λ_v-calculus requires a notion of *value* v. These are defined in the usual way, by

$$v ::= x \mid \lambda x.t$$

consisting of variables and λ-abstractions.

Computation in the λ_v-calculus is through the β_v-reduction rule

$$(\lambda x.t)v \quad \longmapsto_v \quad \{v/x\}t\ .$$

When an abstraction $\lambda x.t$ is applied to a value v then substitute v for x in the body t. The *reduction relation* (also denoted $\longmapsto_v$) is the smallest that satisfies the following rules

$$\frac{}{(\lambda x.t)v \longmapsto_v \{v/x\}t} \qquad \frac{s \longmapsto_v s'}{s\ t \longmapsto_v s'\ t} \qquad \frac{t \longmapsto_v t'}{s\ t \longmapsto_v s\ t'}\ .$$

The transitive closure of the reduction relation is denoted $\longmapsto_v^*$ though the star may be elided if it is obvious from the context.

The more generous operational semantics for the λ-calculus allows any term to be the argument when defining β-reduction. Thus the more generous β-reduction rule is

$$(\lambda x.s)t \quad \longmapsto \quad \{t/x\}s$$

where t is any term of the λ-calculus. The reduction relation $\longmapsto$ and the transitive closure thereof $\longmapsto^*$ are obvious adaptations from those for the λ_v-calculus. Observe that any reduction $\longmapsto_v$ of λ_v-calculus is also a reduction $\longmapsto$ of λ-calculus.

3.2 Traditional Combinatory Logic

A *combinatory calculus* is given by a finite collection $\mathcal{O}$ of *operators* (meta-variable O) that are used to define the $\mathcal{O}$-*combinators* (meta-variables M, N, X, Y, Z) built from these by application

$$M, N ::= O \mid MN\ .$$

Syntactic equality of combinators will be denoted by $\equiv$. The *$\mathcal{O}$-combinatory calculus* or *$\mathcal{O}$-calculus* is given by the combinators plus their reduction rules.

Traditional combinatory logic can be represented by two combinators S and K [8] so the SK-calculus has *reduction rules*

$$\begin{aligned} SMNX &\longmapsto MX(NX) \\ KXY &\longmapsto X \ . \end{aligned}$$

The combinator $SMNX$ duplicates X as the argument to both M and N. The combinator KXY eliminates Y and returns X. The *reduction relation* $\longmapsto$ is as for λ-calculus.

Although this is sufficient to provide a direct account of functions in the style of λ-calculus, an alternative is to consider the representation of arbitrary computable functions that act upon combinators.

A *symbolic function* is defined to be an n-ary partial function $\mathcal{G}$ of some combinatory logic, i.e. a function of the combinators that preserves their equality, as determined by the reduction rules. That is, if $X_i = Y_i$ for $1 \leq i \leq n$ then $\mathcal{G}(X_1, X_2, \ldots, X_n) = \mathcal{G}(Y_1, Y_2, \ldots, Y_n)$ if both sides are defined. A symbolic function is *restricted* to a set of combinators, e.g. the normal forms, if its domain is within the given set.

A combinator G in a calculus *represents* $\mathcal{G}$ if

$$GX_1 \ldots X_n = \mathcal{G}(X_1, \ldots, X_n)$$

whenever the right-hand side is defined. For example, the symbolic functions $\mathcal{S}(X_1, X_2, X_3) = X_1X_3(X_2X_3)$ and $\mathcal{K}(X_1, X_2) = X_1$ are represented by S and K, respectively, in SK-calculus. Consider the symbolic function $\mathcal{I}(X) = X$. In SKI-calculus where I has the rule $IY \longmapsto Y$ then $\mathcal{I}$ is represented by I. In both SKI-calculus and SK-calculus, $\mathcal{I}$ is represented by any combinator of the form SKX since $SKXY = KY(XY) = Y$. For convenience define the *identity combinator* I in SK-calculus to be SKK.

3.3 Relations

One of the goals of combinatory logic is to give an equational account of variable binding and substitution, particularly as it appears in λ-calculus. In order to represent λ-abstraction, it is necessary to have some variables to work with. Given $\mathcal{O}$ as before, define the *$\mathcal{O}$-terms* by

$$M, N ::= x \mid O \mid MN$$

where x is as in λ-calculus. Free variables, substitutions, and symbolic computations are defined just as for $\mathcal{O}$-calculus.

Given a variable x and term M define a symbolic function $\mathcal{G}$ on terms by

$$\mathcal{G}(X) = \{X/x\}M \ .$$

Note that if M has no free variables other than x then $\mathcal{G}$ is also a symbolic computation of the combinatory logic. If every such function $\mathcal{G}$ on $\mathcal{O}$-combinators is representable then the $\mathcal{O}$-combinatory logic is *combinatorially complete* in the sense of Curry [8, p. 5].

Given S and K then $\mathcal{G}$ above can be represented by a term $\lambda^* x.M$ given by

$$
\begin{array}{ll}
\lambda^* x.x = I & \lambda^* x.O = KO \\
\lambda^* x.y = Ky \quad \text{if } y \neq x & \lambda^* x.MN = S(\lambda^* x.M)(\lambda^* x.N)\ .
\end{array}
$$

The following lemma is a central result of combinatory logic [8] and Theorem 2.3 of [19]. This is sufficient to show there is a homomorphism from λ_v-calculus to any combinatory calculus that represents S and K.

Lemma 1. *Any combinatory calculus that is able to represent S and K is combinatorially complete.*

Theorem 1. *There is a homomorphism from λ-calculus into SK-calculus.*

There is a standard translation from SK-calculus into λ-calculus that preserves reduction and supports the following lemma [8, 3].

Lemma 2 (Theorem 2.3.3 of [11]). *Translation from SK-calculus to λ-calculus preserves the reduction relation.*

Theorem 2. *There is a homomorphism from SK-calculus into λ-calculus.*

Although the top left corner of the computation square is populated by λ_v-calculus, the arrows out allow for either λ_v-calculus or SK-calculus to be used. Indeed, the homomorphisms in both directions between λ-calculus and SK-calculus allow these two calculi to be considered equivalent.

4 Sequential Intensional Computation

Intuitively intensional functions are more expressive than merely extensional functions, however populating the top right corner of the computation square requires more formality than intuition. The cleanest account of this is by considering combinatory logic.

Even in SK-calculus there are Turing-computable functions defined upon the combinators that cannot be represented within SK-calculus. For example, consider the function that reduces any combinator of the form SKX to X. Such a function cannot be represented in SK-calculus, or λ-calculus, as all combinators of the form SKX represent the identity function. However, such a function is Turing-computable and definable upon the combinators. This is an example of a more general problem of *factorising* combinators that are both applications and stable under reduction.

Exploiting this factorisation is SF-calculus [19] that is able to support intensional functions on combinators including a structural equality of normal forms. Thus SF-calculus sits at the top right hand corner of the computation square. The arrow across the top of the square is formalised by showing a homomorphism from SK-calculus into SF-calculus. The lack of a converse has been proven by showing that the intensionality of SF-calculus cannot be represented within SK-calculus, or λ-calculus [19].

4.1 Symbolic Functions

Symbolic functions need not be merely extensional, indeed it is possible to define symbolic functions that consider the structure of their arguments. Observe that each operator O has an *arity* given by the minimum number of arguments it requires to instantiate a rule. Thus, K has arity 2 while S has arity 3. A *partially applied operator* is a combinator of the form $OX_1 \ldots X_k$ where k is less than the arity of O. An operator with a positive arity is an *atom* (meta-variable A). A partially applied operator that is an application is a *compound*. Hence, the partially applied operators of SK-calculus are the atoms S and K, and the compounds SM, SMN and KM for any M and N.

Now define a *factorisation function* $\mathcal{F}$ on combinators by

$$\begin{array}{rcll} \mathcal{F}(A,M,N) & \longmapsto & M & \text{if } A \text{ is an atom} \\ \mathcal{F}(XY,M,N) & \longmapsto & NXY & \text{if } XY \text{ is a compound.} \end{array}$$

Lemma 3 (Theorem 3.2 of [19]). *Factorisation of SK-combinators is a symbolic computation that is not representable within SK-calculus.*

4.2 *SF*-calculus

When considering intensionality in a combinatory logic it is tempting to specify a factorisation combinator F as a representative for $\mathcal{F}$. However, $\mathcal{F}$ is defined using partially applied operators, which cannot be known until all reduction rules are given, including those for F. This circularity of definition is broken by beginning with a syntactic characterisation of the combinators that are to be factorable.

The SF-calculus [19] has *factorable forms* given by $S \mid SM \mid SMN \mid F \mid FM \mid FMN$ and *reduction rules*

$$\begin{array}{rcll} SMNX & \longmapsto & MX(NX) & \\ FOMN & \longmapsto & M & \text{if } O \text{ is } S \text{ or } F \\ F(XY)MN & \longmapsto & NXY & \text{if } XY \text{ is a factorable form.} \end{array}$$

The expressive power of SF-calculus subsumes that of SK-calculus since K is here defined to be FF and I is defined to be SKK as before.

Lemma 4 (Theorem 5.2.3 of [11]). *There is a homomorphism from SK-calculus into SF-calculus.*

Theorem 3 (Corollary 5.2.4 of [11]). *There is a homomorphism from λ_v-calculus to SF-calculus.*

Theorem 4. *There is no homomorphism from SF-calculus to λ_v-calculus.*

Proof. Observe that $FyS(FF(S(FF)(FF)))$ with y replaced by a combinator of the form SKX reduces to X. By Lemma 3 it is impossible to represent this reduction in SK-calculus and thus reduction cannot be preserved.

This completes the top edge of the computation square by showing that SF-calculus subsumes λ_v-calculus and that the subsumption is irreversible. Indeed, these results hold for λ-calculus [11, Theorem 5.2.6] and SK-calculus (by Lemma 3) as well.

5 Concurrent Extensional Computation

The bottom left corner of the computation square considers extensional concurrent computation, here defined to be extensional process calculi that subsume λ-calculus. The π-calculus [25] holds a pivotal rôle amongst process calculi due to popularity, being the first to represent topological changes, and subsuming λ_v-calculus [23]. Note that although there are many π-calculi, the one here is that used by Milner so as to more easily exploit previous results [23] (and here augmented with a success process $\surd$).

The processes for the π-calculus are given as follows and exploit a class of names (denoted $m, n, x, y, z, \ldots$ similar to variables in the λ-calculus):

$$P \quad ::= \quad \mathbf{0} \ | \ P|P \ | \ !P \ | \ (\nu a)P \ | \ a(b).P \ | \ \overline{a}\langle b\rangle.P \mid \surd\ .$$

The names of the π-calculus are used for channels of communication and for information being communicated. The *free names* of a process $\mathsf{fn}(P)$ are as usual. *Substitutions* in the π-calculus are partial functions that map names to names, with domain, range, free names, names, and avoidance, all straightforward adaptations from substitutions of the λ-calculus. The application of a substitution to a process is defined in the usual manner. Issues where substitutions must avoid restricted or input names are handled by α-conversion $=_\alpha$ that is the congruence relation defined in the usual manner. The general *structural equivalence relation* $\equiv$ is defined by:

$$\begin{gathered}
P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
!P \equiv P \mid !P \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \qquad (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \\
P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) \ \text{ if } n \notin \mathsf{fn}(P)
\end{gathered}$$

The π-calculus has one *reduction rule* given by

$$a(b).P \mid \overline{a}\langle c\rangle.Q \quad \longmapsto \quad \{c/b\}P \mid Q\ .$$

The reduction rule is then closed under parallel composition, restriction and structural equivalence to yield the reduction relation $\longmapsto$ as follows:

$$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q} \qquad \frac{P \longmapsto P'}{(\nu n)P \longmapsto (\nu n)P'} \qquad \frac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'}\ .$$

The reflexive, transitive closure of $\longmapsto$ is denoted $\Longmapsto$.

Now that the π-calculus and process calculus concepts are recalled, it remains to demonstrate that Milner's encoding [23] can meet the criteria for a parallel encoding.

As the β_v-reduction rule depends upon the argument being a value the translation into π-calculus must be able to recognise values. Thus, Milner defines the following

$$[\![y := \lambda x.t]\!] \stackrel{\text{def}}{=} !y(w).w(x).w(c).[\![t]\!]_c \qquad [\![y := x]\!] \stackrel{\text{def}}{=} !y(w).\overline{x}\langle w\rangle\ .$$

Also the following translation of λ_v-terms

$$\begin{aligned}
[\![v]\!]_c &\stackrel{\text{def}}{=} (\nu y)\overline{c}\langle y\rangle.[\![y := v]\!] && y \text{ not free in } v \\
[\![s\ t]\!]_c &\stackrel{\text{def}}{=} (\nu q)(\nu r)(\mathsf{ap}(c, q, r) \mid [\![s]\!]_q \mid [\![t]\!]_r) \\
\mathsf{ap}(p, q, r) &\stackrel{\text{def}}{=} q(y).(\nu v)\overline{y}\langle v\rangle.r(z).\overline{v}\langle z\rangle.\overline{v}\langle p\rangle\ .
\end{aligned}$$

Lemma 5. *The translation* $[\![\cdot]\!]_c$ *preserves and reflects reduction. That is:*

1. *If* $s \longmapsto_v t$ *then* $[\![s]\!]_c \Longmapsto\simeq [\![t]\!]_c$*;*
2. *if* $[\![s]\!]_c \longmapsto Q$ *then there exists* Q' *and* s' *such that* $Q \Longmapsto Q'$ *and* $Q' \simeq [\![s']\!]_c$ *and either* $s \longmapsto_v s'$ *or* $s = s'$*.*

Theorem 5. *The translation* $[\![\cdot]\!]_c$ *is a parallel encoding from* λ_v*-calculus to* π*-calculus.*

There is some difficulty in attempting to define the analogue of a parallel encoding or homomorphism from a language with a parallel composition operator into a language without. However, this difficulty can be avoided by observing that any valid encoding, parallel encoding, or homomorphism must preserve reduction. Reduction preservation can then be exploited to show when an encoding is impossible. Here this is by exploiting Theorem 14.4.12 of Barendregt [3], showing that λ-calculus is unable to render concurrency or support concurrent computations.

Theorem 6. *There is no reduction preserving encoding of* π*-calculus into* λ*-calculus.*

6 Concurrent Intensional Computation

Intensionality in sequential computation yields greater expressive power so it is natural to consider intensional concurrent computation. Intensionality in CPC is supported by a generalisation of pattern-matching to symmetric *pattern-unification* that provides the basis for defining interaction.

6.1 Concurrent Pattern Calculus

The *patterns* (meta-variables $p, p', p_1, q, q', q_1, \ldots$) are built using a class of *names* familiar from π-calculus and have the following forms

$$p ::= \lambda x \mid x \mid \ulcorner x \urcorner \mid p \bullet p$$

Binding names λx denote an input sought by the pattern. Variable names x may be output or tested for equality. Protected names $\ulcorner x \urcorner$ can only be tested for equality. A compound combines two patterns p and q, its *components*, into a pattern $p \bullet q$ and is left associative. The *atoms* are patterns that are not compounds and the atoms x and $\ulcorner x \urcorner$ are defined to *know* x. The binding names of a pattern must be pairwise distinct.

A *communicable* pattern contains no binding or protected names. Given a pattern p, the binding names $\mathsf{bn}(p)$, variable names $\mathsf{vn}(p)$, and protected names $\mathsf{pn}(p)$, are as expected, with the free names $\mathsf{fn}(p)$ being the union of variable and protected names.

A *substitution* σ (also denoted $\sigma_1, \rho, \rho_1, \theta, \theta_1, \ldots$) is a partial function from names to communicable patterns. Otherwise substitutions and their properties are familiar from earlier sections and are applied to patterns in the obvious manner. (Observe that protection can be extended to a communicable pattern by $\ulcorner p \bullet q \urcorner = \ulcorner p \urcorner \bullet \ulcorner q \urcorner$ in the application of a substitution to a protected name.)

The *symmetric matching* or *unification* $\{p \parallel q\}$ of two patterns p and q attempts to unify p and q by generating substitutions upon their binding names. When defined, the result is some pair of substitutions whose domains are the binding names of p and of q, respectively. The rules to generate the substitutions are:

$$\begin{array}{rll}
\{x \,\|\, x\} = \{x \,\|\, \ulcorner x\urcorner\} \;=\; \{\ulcorner x\urcorner \,\|\, x\} = \{\ulcorner x\urcorner \,\|\, \ulcorner x\urcorner\} & \stackrel{\text{def}}{=} (\{\},\{\}) & \\
\{\lambda x \,\|\, q\} & \stackrel{\text{def}}{=} (\{q/x\},\{\}) & \text{if } q \text{ is communicable} \\
\{p \,\|\, \lambda x\} & \stackrel{\text{def}}{=} (\{\},\{p/x\}) & \text{if } p \text{ is communicable} \\
\{p_1 \bullet p_2 \,\|\, q_1 \bullet q_2\} & \stackrel{\text{def}}{=} (\sigma_1 \cup \sigma_2 \,,\, \rho_1 \cup \rho_2) & \text{if } \{p_i \,\|\, q_i\} = (\sigma_i, \rho_i) \text{ for } i \in \{1,2\}
\end{array}$$

Two atoms unify if they know the same name. A binding name unifies with any communicable pattern to produce a binding for its underlying name. Two compounds unify if their corresponding components do; the resulting substitutions are given by taking unions of those produced by unifying the components. Otherwise the patterns cannot be unified and the matching is undefined.

The processes of CPC are the same as for π-calculus except for the input and output replaced by the *case* $p \to P$ with pattern p and body P. A case with the null process as the body $p \to \mathbf{0}$ may also be written p when no ambiguity may occur.

The free names of processes, denoted $\mathsf{fn}(P)$, are defined as usual for all the traditional primitives and $\mathsf{fn}(p \to P) \quad = \quad \mathsf{fn}(p) \cup (\mathsf{fn}(P) \backslash \mathsf{bn}(p))$ for the case. As expected the binding names of the pattern bind their free occurrences in the body. The application σP of a substitution σ to a process P is defined in the usual manner to avoid name capture. For cases this ensures that substitution avoids the binding names in the pattern: $\sigma(p \to P) = (\sigma p) \to (\sigma P)$ if σ avoids $\mathsf{bn}(p)$. Renaming of a case is handled through α-conversion, $=_\alpha$, that is the congruence relation generated by the following axiom $p \to P =_\alpha (\{\lambda y/\lambda x\}p) \to (\{y/x\}P)$ when $x \in \mathsf{bn}(p), y \notin \mathsf{fn}(P) \cup \mathsf{bn}(p)$. Renaming of a restriction is as usual. The renaming of a binding name (e.g. by $\{\lambda y/\lambda x\}$) is also as expected with the usual restrictions. The general *structural equivalence relation* $\equiv$ is defined just as in π-calculus.

CPC has one *interaction axiom* given by

$$(p \to P) \mid (q \to Q) \quad \longmapsto \quad (\sigma P) \mid (\rho Q) \qquad \text{if } \{p \,\|\, q\} = (\sigma, \rho)\ .$$

It states that if the unification of two patterns p and q is defined and generates (σ, ρ), then apply the substitutions σ and ρ to the bodies P and Q, respectively. If the matching of p and q is undefined then no interaction occurs. The interaction rule is then closed under parallel composition, restriction and structural equivalence in the usual manner. The reflexive and transitive closure of $\longmapsto$ is denoted $\Longmapsto$. Finally, the reference behavioural equivalence relation $\sim$ for CPC is already well detailed [11, 12, 14].

6.2 Completing the Square

Support for both intensionality and concurrency places CPC at the bottom right corner of the computation square. This section shows how SF-calculus and π-calculus can both be subsumed by CPC, and thus completes the computation square.

Down the right side of the square there is a parallel encoding from SF-calculus into CPC that also maps the combinators S and F to reserved names S and F, respectively. The impossibility of finding a parallel encoding of CPC into SF-calculus is proved in the same manner as the relation between λ_v-calculus and π-calculus. Interestingly, in contrast with the parallel encoding of λ-calculus into π-calculus, the parallel encoding of SF-calculus into CPC does *not* fix a reduction strategy for SF-calculus. This is achieved by exploiting the intensionality of CPC to directly encode the reduction rules for SF-calculus into an SF-reducing process, or SF-machine. In turn, this process can then operate on translated combinators and so support reduction and rewriting.

The square is completed by showing a homomorphism from π-calculus into CPC, and by showing that there cannot be any homomorphism (or indeed a more general valid encoding) from CPC into π-calculus.

***SF*-calculus.** The SF-calculus combinators can be easily encoded into patterns by defining the *construction* $(\!|\cdot|\!)$, exploiting reserved names S and F, as follows

$$(\!|S|\!) \stackrel{\text{def}}{=} S \qquad (\!|F|\!) \stackrel{\text{def}}{=} F \qquad (\!|MN|\!) \stackrel{\text{def}}{=} (\!|M|\!) \bullet (\!|N|\!)\ .$$

Observe that the first two rules map the operators to the same names. The third rules maps application to a compound of the components M and N.

By representing SF-calculus combinators in the pattern of a CPC case, the reduction can then be driven by defining cases that recognise a reducible structure and perform the appropriate operations. The reduction rules can be captured by matching on the structure of the left hand side of the rule and reducing to the structure on the right. So (considering each possible instance for the F reduction rules) they can be encoded by cases as follows.

$$\begin{aligned} S \bullet \lambda m \bullet \lambda n \bullet \lambda x &\to m \bullet x \bullet (n \bullet x) \\ F \bullet S \bullet \lambda m \bullet \lambda n &\to m \\ F \bullet F \bullet \lambda m \bullet \lambda n &\to m \\ &\dots \\ F \bullet (F \bullet \lambda p \bullet \lambda q) \bullet \lambda m \bullet \lambda n &\to n \bullet (F \bullet p) \bullet q\ . \end{aligned}$$

These processes capture the reduction rules, matching the pattern for the left hand side and transforming it to the structure on the right hand side. Of course these process do not capture the possibility of reduction of a sub-combinator, so further rules are required. Rather than detail them all, consider the example of a reduction $MNOP \longmapsto MN'OP$ that can be captured by

$$\lambda m \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x) \bullet \lambda o \bullet \lambda p \to u \bullet v \bullet w \bullet x \to \lambda z \to m \bullet z \bullet o \bullet p$$

This process unifies with a combinator $MXOP$ where X is reducible (observable from the structure), here binding the components of X to four names u, v, w and x. These four names are then shared as a pattern, which can then be unified with another process that can perform the reduction. The result will then (eventually) unify with λz and be substituted back into $m \bullet z \bullet o \bullet p$ to complete the reduction.

To exploit these processes in constructing a parallel encoding requires the addition of a name, used like a channel, to control application. Thus, prefix each pattern that

$$
\begin{aligned}
&!\lambda c \bullet (S \bullet \lambda m \bullet \lambda n \bullet \lambda x) \to c \bullet (m \bullet x \bullet (n \bullet x))\\
&\mid\ !\lambda c \bullet (F \bullet S \bullet \lambda m \bullet \lambda n) \to c \bullet m\\
&\mid\ !\lambda c \bullet (F \bullet F \bullet \lambda m \bullet \lambda n) \to c \bullet m\\
&\mid\ !\lambda c \bullet (F \bullet (S \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet S \bullet q)\\
&\mid\ !\lambda c \bullet (F \bullet (F \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet F \bullet q)\\
&\mid\ !\lambda c \bullet (F \bullet (S \bullet \lambda p \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet (S \bullet p) \bullet q)\\
&\mid\ !\lambda c \bullet (F \bullet (F \bullet \lambda p \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet (F \bullet p) \bullet q)\\
&\mid\ !\lambda c \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x \bullet \lambda y)\\
&\qquad \to (\nu d)d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (z \bullet y)\\
&\mid\ !\lambda c \bullet (\lambda m \bullet \lambda n \bullet \lambda o \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x))\\
&\qquad \to (\nu d)d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (m \bullet n \bullet o \bullet z)\\
&\mid\ !\lambda c \bullet (\lambda m \bullet \lambda n \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x) \bullet \lambda p)\\
&\qquad \to (\nu d)d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (m \bullet n \bullet z \bullet p)\\
&\mid\ !\lambda c \bullet (\lambda m \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x) \bullet \lambda o \bullet \lambda p)\\
&\qquad \to (\nu d)d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (m \bullet z \bullet o \bullet p)
\end{aligned}
$$

Fig. 1. The SF-reducing process $\mathcal{R}$.

matches the structure of an SF-combinator with a binding name λc and add this to the result, e.g. $\lambda c \bullet (F \bullet S \bullet \lambda m \bullet \lambda n) \to c \bullet m$. Now the processes that handle each possible reduction rule can be placed under a replication and in parallel composition with each other. This yields the SF-reducing process $\mathcal{R}$ as shown in Figure 1 where the last four rules capture reduction of sub-combinators.

The translation $[\![\cdot]\!]_c$ from SF-combinators into CPC processes is here parametrised by a name c and combines application with a process $\mathsf{ap}(c, m, n)$. This is similar to Milner's encoding from λ_v-calculus into π-calculus and allows the parallel encoding to exploit compositional encoding of sub-terms as processes and thus parallel reduction, while preventing confusion of application.

The translation $[\![\cdot]\!]_c$ of SF-combinators into CPC, exploiting the SF-reducing process $\mathcal{R}$ and reserved names S and F, is defined as follows:

$$
\begin{aligned}
[\![S]\!]_c &\stackrel{\text{def}}{=} c \bullet S \mid \mathcal{R}\\
[\![F]\!]_c &\stackrel{\text{def}}{=} c \bullet F \mid \mathcal{R}\\
[\![MN]\!]_c &\stackrel{\text{def}}{=} (\nu m)(\nu n)(\mathsf{ap}(c, m, n) \mid [\![M]\!]_m \mid [\![N]\!]_n)\\
\mathsf{ap}(c, m, n) &\stackrel{\text{def}}{=} m \bullet \lambda x \to n \bullet \lambda y \to c \bullet (x \bullet y) \mid \mathcal{R}\ .
\end{aligned}
$$

The following lemmas are at the core of the operational correspondence and divergence reflection components of the proof of valid encoding, similar to Milnder's Theorem 7.7 [23]. Further, it provides a general sense of how to capture the reduction of combinatory logics or similar rewrite systems. (Note that the results exploit that $\mathcal{R} \mid \mathcal{R} \sim \mathcal{R}$ to remove redundant copies of $\mathcal{R}$ [11, Theorem 8.7.2].)

Lemma 6. *Given an SF-combinator M the translation $[\![M]\!]_c$ has a reduction sequence to a process of the form $c \bullet (\!|M|\!) \mid \mathcal{R}$.*

Lemma 7 (Theorem 7.1.2 of [11]). *Given an SF-combinator M the translation $[\![M]\!]_c$ preserves reduction.*

Lemma 8. *The translation* $[\![\cdot]\!]_c$ *preserves and reflects reduction. That is:*

1. *If* $M \longmapsto N$ *then* $[\![M]\!]_c \Longmapsto\simeq [\![N]\!]_c$*;*
2. *if* $[\![M]\!]_c \longmapsto Q$ *then there exists* Q' *and* N *such that* $Q \Longmapsto Q'$ *and* $Q' \simeq [\![N]\!]_c$ *and either* $M \longmapsto N$ *or* $M = N$*.*

Theorem 7. *The translation* $[\![\cdot]\!]_c$ *is a parallel encoding from* SF*-calculus to CPC.*

The lack of an encoding of CPC (or even π-calculus) into SF-calculus can be proved in the same manner as Theorem 6 for showing no encoding of π-calculus into λ-calculus.

Theorem 8. *There is no reduction preserving encoding from CPC into* SF*-calculus.*

It may appear that the factorisation operator F adds some expressiveness that could be used to capture the parallel-or function g. Perhaps use F to switch on the result of the first function so that (assuming true is some operator $\mathtt{T}$ then) $g\ x\ y$ is represented by $Fx\mathtt{T}(K(Ky))$ that reduces to $\mathtt{T}$ when $x = \mathtt{T}$ and to $K(Ky)MN \Longmapsto y$ when $x = MN$ that somehow is factorable but not terminating. However, this kind of attempt is equivalent to exploiting factorisation to detect termination and turns out to be paradoxical as demonstrated in the proof of Theorem 5.1 of [19].

This completes the arrow down the right side of the computation square. The rest of this section discusses some properties of translations and the diagonal from the top left to the bottom right corner of the square.

Observe that the parallel encoding from SF-calculus into CPC does not require the choice of a reduction strategy, unlike Milner's encodings from λ-calculus into π-calculus. The structure of patterns and peculiarities of pattern-unification allow the reduction relation to be directly rendered by CPC. In some sense this is similar to encoding the SF-combinators onto the tape of a Turing machine, the pattern $(\!|\cdot|\!)$, and providing another process to be the state that reads the tape and performs operations upon it, the SF-reducing process $\mathcal{R}$. This approach can also be adapted in a straightforward manner to support a parallel encoding of SK-calculus into CPC, that like the encoding of SF-calculus does not fix a reduction strategy.

Theorem 9. *There is a translation* $[\![\cdot]\!]_c$ *that is a parallel encoding from* SK*-calculus into CPC.*

The translation from SF-calculus to CPC presented here is designed to map application to parallel composition (with some restriction and process R) so as to meet the parallelisation and compositionality criteria for a parallel encoding. However, the construction $(\!|\cdot|\!)$ can be used to provide a cleaner translation if these are not required. Consider an alternative translation $[\![\cdot]\!]^c$ parametrised by a name c as usual and defined by $[\![M]\!]^c \stackrel{\text{def}}{=} c \bullet (\!|M|\!) \mid \mathcal{R}$. Such a translation still supports name invariance, operational correspondence (up to structural equivalence $\equiv$ instead of weak behavioural equivalence $\simeq$), and divergence reflection.

π-calculus. Across the bottom of the computation square there is a homomorphism from π-calculus into CPC. The converse separation result can be proved multiple ways [13, 11, 14].

The translation $[\![\cdot]\!]$ from π-calculus into CPC is homomorphic on all process forms except for the input and output which are translated as follows:

$$
\begin{aligned}
[\![a(b).P]\!] &\stackrel{\text{def}}{=} a \bullet \lambda b \bullet \mathsf{in} \to [\![P]\!] \\
[\![\overline{a}\langle b\rangle.P]\!] &\stackrel{\text{def}}{=} a \bullet b \bullet \lambda x \to [\![P]\!] \qquad x \text{ not free in } P.
\end{aligned}
$$

Here in is any name, a symbolic name is used for clarity but no result relies upon this choice. The fresh name x is used to prevent the introduction of new reductions due to CPC's symmetric matching.

Lemma 9 (Corollary 7.2.3 of [11]). *The translation $[\![\cdot]\!]$ from π-calculus into CPC is a valid encoding.*

Theorem 10. *There is a homomorphism from π-calculus into CPC.*

Thus the translation provided above is a homomorphism from π-calculus into CPC. Now consider the converse separation result.

Lemma 10 (Theorem 7.2.5 of [11]). *There is no valid encoding of CPC into π-calculus.*

Theorem 11. *There is no homomorphism from CPC into π-calculus.*

7 Conclusions and Future Work

This work illustrates that there are increases in expressive power by shifting along two dimensions: from extensional to intensional, and from sequential to concurrent. This is best illustrated by the computation square that relates λ_v-calculus, SF-calculus, π-calculus, and CPC as follows

$$
\begin{tikzcd}
\lambda_v\text{-calculus} \arrow[r] \arrow[d, dashed] & SF\text{-calculus} \arrow[d, dashed] \\
\pi\text{-calculus} \arrow[r] & \text{concurrent pattern calculus}
\end{tikzcd}
$$

where the left side is extensional, the right side intensional, the top side sequential, and the bottom side concurrent. The horizontal arrows are homomorphisms that map application/parallel composition to itself, and preserve and reflect reduction. The vertical arrows are parallel encodings that map application to parallel composition (with some extra machinery), and preserve and reflect reduction. Further, there are no reverse arrows as each arrow signifies an increase in expressive power.

Such a square identifies relations that are more general than simply the choice of calculi here. The top left corner could be populated by λ_v-calculus or λ_l-calculus with minimal changes to the proofs. Alternatively, choosing λ-calculus or SK-calculus may also hold, although a parallel encoding into π-calculus requires some work. The top right corner could be populated by any of the structure complete combinatory logics without much effort [19, 11]. It may also be possible to place a pattern calculus [20, 18], at the top right. The bottom left corner is also open to many other calculi: monadic/polyadic synchronous/asynchronous π-calculus could replace π-calculus with no significant changes to the results [11, 14]. Similarly there are, and will be, other process calculi that can take the place of CPC at the bottom right. For Spi calculus [1] an encoding of SF-calculus is delicate due to correctly handling reduction and not introducing infinite reductions or blocking on Spi calculus primitives and reductions. For Psi calculus [4] the encoding can be achieved very similarly to CPC, although arguably the implicit computation component of Psi calculus could simply allow for SF-calculus with the rest being moot. Although multiple process calculi may populate the bottom right hand corner, the elegance of CPC's intensionality is illustrated by the construction $(\!| \cdot |\!)$ for combinatory logics.

Related Work

The choice of relations here is influenced by existing approaches. Homomorphisms in the sequential setting are standard [8, 3, 10]. Valid encodings are popular [15–17, 21, 22, 30] albeit not the only approach as other ways to relate process calculi are also used that vary on the choice to map parallel composition to parallel composition (i.e. the homomorphism requirement here) [27, 6, 9, 26, 30]. Since the choice here is to build on prior results, valid encodings are the obvious basis for relating π-calculus and CPC, but no doubt this could be formalised under different criteria. Finally, the definition of parallel encodings here is to exploit the existing encodings in the literature and to be similar to valid encodings/homomorphisms. However, other approaches are possible as in [23, 28] and doubtless many more as encoding λ-calculus into process calculi is common [5, 25, 7, 24].

The separation results here build upon results already in the literature. For showing the inability to encoding concurrent languages into sequential, the work of Abramsky [2] and Plotkin [29] can also be considered. The impossibility of encoding CPC into π-calculus can be proved by using matching degree or symmetry [11, proofs for Theorem 7.2.5].

Future Work

Future work may proceed along several directions. The techniques used to encode SF-calculus into CPC can be generalised for any combinatory logic, indeed it is likely a general result can be proved for all similar rewrite systems. This approach could also be used to clarify the relation between Turing machines and process calculi directly, rather than through λ-calculus. Another path of exploration is to generalise the account of intensionality in concurrency with full results in a general manner, this would include formalising the intensionality (or lack of) of Spi calculus, Psi calculus, and other popular process calculi.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1 – 70, 1999.

2. S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.
3. H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science Publishers B.V., 1985.
4. J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.
5. G. Berry and G. Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM.
6. N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of linda coordination primitives. *Inf. Comput.*, 156(1-2):90–121, 2000.
7. L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.
8. H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.
9. R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of klaim-based calculi. *Theor. Comput. Sci.*, 356(3):387–421, May 2006.
10. M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(13):35 – 75, 1991.
11. T. Given-Wilson. *Concurrent Pattern Unification*. PhD thesis, University of Technology, Sydney, Australia, 2012.
12. T. Given-Wilson and D. Gorla. Pattern matching and bisimulation. In *Proc. of COORDINATION*, volume 7890 of *LNCS*, pages 60–74. Springer, 2013.
13. T. Given-Wilson, D. Gorla, and B. Jay. Concurrent pattern calculus. In *Proc. of IFIP-TCS*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 244–258. Springer, 2010.
14. T. Given-Wilson, D. Gorla, and B. Jay. A concurrent pattern calculus. *To appear in Logical Methods in Computer Science*, 2014. `http://www.progsoc.org/~sanguinev/files/20140414-journal-cpc.pdf`.
15. D. Gorla. Comparing communication primitives via their relative expressive power. *Information and Computation*, 206(8):931–952, 2008.
16. D. Gorla. A taxonomy of process calculi for distribution and mobility. *Distributed Computing*, 23(4):273–299, 2010.
17. D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010.
18. B. Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.
19. B. Jay and T. Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.
20. B. Jay and D. Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.
21. I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness of polyadic and synchronous communication in higher-order process calculi. In *Proc. of ICALP*, volume 6199 of *LNCS*, pages 442–453. Springer, 2010.
22. I. Lanese, C. Vaz, and C. Ferreira. On the expressive power of primitives for compensation handling. In *Proc. of ESOP*, volume 6012 of *LNCS*, pages 366–386. Springer, 2010.
23. R. Milner. Functions as processes. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 167–180, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
24. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

25. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.
26. L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sum types. In *EXPRESS*, pages 121–135, 2010.
27. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. *CoRR*, cs.PL/9809008, 1998.
28. J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of LICS*, pages 176–185. IEEE Computer Society, 1998.
29. G. Plotkin. Full abstraction, totality and pcf. *Math. Structures Comput. Sci*, 9, 1997.
30. R. J. van Glabbeek. Musings on encodings and expressiveness. In *Proc. of EXPRESS/SOS*, volume 89 of *EPTCS*, pages 81–98, 2012.

Appendix

This appendix provides greater detail for proofs omitted in the body of the paper.

Lemma 11. *For all terms M and N and variables x there is a reduction $(\lambda^* x.M)\ N \longmapsto^* \{N/x\}M$.*

Proof. The proof straightforward is by induction on the structure of the combinator M.

Proof (Lemma 1). Given $\mathcal{G}(X) = \{X/x\}M$ as above define G to be $\lambda^* x.M$ and apply Lemma 11.

Proof (Theorem 1). Straightforward by definition of the translation and Lemma 1.

Proof (Theorem 2). Straightforward by definition of the translation and Lemma 2.

Proof (Lemma 5). The first part can be proved by exploiting Milner's Theorem 7.7 [23]. The second is by considering the reduction $[\![\ s\ ]\!]_c \longmapsto Q$ which must arise from the encoding of an application. It is then straightforward to show that either: the reductions $Q \Longmapsto Q'$ correspond only to translated applications and thus $Q' \simeq [\![\ s\ ]\!]_c$; or the reductions are due to a λ_v-abstraction and thus $Q' \simeq [\![\ s'\ ]\!]_c$ and $s \longmapsto_v s'$.

Proof (Theorem 5). Compositionality, parallelisation, and name invariance hold by construction. Operational correspondence follows from Lemma 5. Divergence reflection can be proved by observing that the only reductions introduced in the translation that do not correspond to reductions in the source language are from translated applications, and these are bounded by the size of the source term.

Proof (Theorem 6). Define the parallel-or function and show that it can be represented in π-calculus. The parallel-or function is a function $g(x, y)$ that satisfies the following three rules $g(\bot, \bot) \longmapsto^* \bot$ and $g(\mathtt{T}, \bot) \longmapsto^* \mathtt{T}$ and $g(\bot, \mathtt{T}) \longmapsto^* \mathtt{T}$ where $\bot$ represents non-termination and $\mathtt{T}$ represents true. Such a function is trivial to encode in π-calculus by $g(n_1, n_2) = G = n_1(x).\overline{m}\langle x\rangle.\mathbf{0} \mid n_2(x).\overline{m}\langle x\rangle.\mathbf{0}$. Consider G in parallel with two processes P_1 and P_2 that output their result on n_1 and n_2, respectively. If either P_1 or P_2 outputs $\mathtt{T}$ then G will also output $\mathtt{T}$ along m. Clearly π-calculus can represent the parallel-or function, and since Barendregt's Theorem 14.4.12 shows that λ-calculus cannot, there cannot be any reduction preserving encoding of π-calculus into λ-calculus.

Proof (Lemma 6). The proof is by induction on the structure of M.

- If M is S or F then $[\![M]\!]_c = c \bullet M \mid \mathcal{R} = c \bullet (\!|M|\!) \mid \mathcal{R}$ and the result is immediate.
- If M is of the form $M_1 M_2$ then $[\![M]\!]_c$ is of the form

$$(\nu m)(\nu n)(m \bullet \lambda x \to n \bullet \lambda y \to c \bullet (x \bullet y) \mid \mathcal{R} \mid [\![M_1]\!]_m \mid [\![M_2]\!]_n)\ .$$

  By two applications of induction, $[\![M_1]\!]_m \Longmapsto m \bullet (\!|M_1|\!) \mid \mathcal{R}$ and also $[\![M_2]\!]_n \Longmapsto n \bullet (\!|M_2|\!) \mid \mathcal{R}$. Now there are reductions as follows:

$$\begin{aligned}
[\![M]\!]_c &\Longmapsto (\nu m)(\nu n)(m \bullet \lambda x \to n \bullet \lambda y \to c \bullet (x \bullet y) \mid \mathcal{R} \\
&\qquad\qquad \mid m \bullet (\!|M_1|\!) \mid n \bullet (\!|M_2|\!)) \\
&\longmapsto (\nu m)(\nu n)(n \bullet \lambda y \to c \bullet ((\!|M_1|\!) \bullet y) \mid \mathcal{R} \mid n \bullet (\!|M_2|\!)) \\
&\longmapsto (\nu m)(\nu n)(c \bullet ((\!|M_1|\!) \bullet (\!|M_2|\!)) \mid \mathcal{R}) \\
&= \ c \bullet (\!|M_1 M_2|\!) \mid \mathcal{R}
\end{aligned}$$

yielding the required result.

Proof (Lemma 8). The first part can be proved by exploiting Lemmas 6 and 7. The second is by considering the reduction $[\![M]\!]_c \longmapsto Q$ which must arise from the encoding of an application. It is then straightforward to show that either: the reductions $Q \Longmapsto Q'$ correspond only to rebuilding the structure as in Lemma 6; or the reductions correspond to a reduction $M \longmapsto N$ and $Q' \simeq [\![N]\!]_c$.

Proof (Theorem 7). Compositionality, parallelisation, and name invariance hold by construction. Operational correspondence follows from Lemma 7. Divergence reflection can be proved by observing that the only reductions introduced in the translation that do not correspond to reductions in the source language are from translated applications, and these are bounded by the size of the source term.

Proof (Theorem 8). The same technique as for Theorem 6 can be used.

Proof (Theorem 9). Straightforward by adapting the techniques used for the translation $[\![\cdot]\!]_c$ from SF-calculus to CPC that is a parallel encoding.

Proof (Theorem 10). By the definition of translation $[\![\cdot]\!]$ and Lemma 9.

Proof (Theorem 11). By Lemma 10.